C1_W4_Assignment

July 23, 2020

1 Assignment 4 - Naive Machine Translation and LSH

You will now implement your first machine translation system and then you will see how locality sensitive hashing works. Let's get started by importing the required functions!

If you are running this notebook in your local computer, don't forget to download the twitter samples and stopwords from nltk.

```
nltk.download('stopwords')
nltk.download('twitter_samples')
```

NOTE: The `Exercise xx` numbers in this assignment ***are inconsistent*** with the `UNQ_Cx` numbers.

1.0.1 This assignment covers the folowing topics:

- Section **??**
  - Section **??**
    - Section **??**
- Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
    - Section **??**
  - Section **??**
    - Section **??**
    - Section **??**
- Section **??**
  - Section **??**
    - Section **??**

In [109]:
```
import pdb
import pickle
import string

import time

import gensim
import matplotlib.pyplot as plt
import nltk
import numpy as np
import scipy
import sklearn
from gensim.models import KeyedVectors
from nltk.corpus import stopwords, twitter_samples
from nltk.tokenize import TweetTokenizer

from utils import (cosine_similarity, get_dict,
                   process_tweet)
from os import getcwd
```

In [110]:
```
# add folder, tmp2, from our local workspace containing pre-downloaded corpora files
nltk.download('stopwords')
nltk.download('twitter_samples')
filePath = f"{getcwd()}/../tmp2/"
nltk.data.path.append(filePath)
```

```
[nltk_data] Downloading package stopwords to /home/jovyan/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package twitter_samples to
[nltk_data]     /home/jovyan/nltk_data...
[nltk_data]   Package twitter_samples is already up-to-date!
```

2 1. The word embeddings data for English and French words

Write a program that translates English to French.

2.1 The data

The full dataset for English embeddings is about 3.64 gigabytes, and the French embeddings are about 629 megabytes. To prevent the Coursera workspace from crashing, we've extracted a subset of the embeddings for the words that you'll use in this assignment.

If you want to run this on your local computer and use the full dataset, you can download the * English embeddings from Google code archive word2vec look for GoogleNews-vectors-negative300.bin.gz * You'll need to unzip the file first. * and the French embeddings from cross_lingual_text_classification. * in the terminal, type (in one line) `curl -o ./wiki.multi.fr.vec https://dl.fbaipublicfiles.com/arrival/vectors/wiki.multi.fr.vec`

Then copy-paste the code below and run it.

```
# Use this code to download and process the full dataset on your local computer

from gensim.models import KeyedVectors

en_embeddings = KeyedVectors.load_word2vec_format('./GoogleNews-vectors-negative300.bin', bina
fr_embeddings = KeyedVectors.load_word2vec_format('./wiki.multi.fr.vec')


# loading the english to french dictionaries
en_fr_train = get_dict('en-fr.train.txt')
print('The length of the english to french training dictionary is', len(en_fr_train))
en_fr_test = get_dict('en-fr.test.txt')
print('The length of the english to french test dictionary is', len(en_fr_train))

english_set = set(en_embeddings.vocab)
french_set = set(fr_embeddings.vocab)
en_embeddings_subset = {}
fr_embeddings_subset = {}
french_words = set(en_fr_train.values())

for en_word in en_fr_train.keys():
    fr_word = en_fr_train[en_word]
    if fr_word in french_set and en_word in english_set:
        en_embeddings_subset[en_word] = en_embeddings[en_word]
        fr_embeddings_subset[fr_word] = fr_embeddings[fr_word]


for en_word in en_fr_test.keys():
    fr_word = en_fr_test[en_word]
    if fr_word in french_set and en_word in english_set:
        en_embeddings_subset[en_word] = en_embeddings[en_word]
```

```
        fr_embeddings_subset[fr_word] = fr_embeddings[fr_word]


pickle.dump( en_embeddings_subset, open( "en_embeddings.p", "wb" ) )
pickle.dump( fr_embeddings_subset, open( "fr_embeddings.p", "wb" ) )
```

The subset of data To do the assignment on the Coursera workspace, we'll use the subset of word embeddings.

```
In [111]: en_embeddings_subset = pickle.load(open("en_embeddings.p", "rb"))
          fr_embeddings_subset = pickle.load(open("fr_embeddings.p", "rb"))
          #print(en_embeddings_subset)
```

Look at the data

- en_embeddings_subset: the key is an English word, and the vaule is a 300 dimensional array, which is the embedding for that word.

```
'the': array([ 0.08007812,  0.10498047,  0.04980469,  0.0534668 , -0.06738281, ....
```

- fr_embeddings_subset: the key is an French word, and the vaule is a 300 dimensional array, which is the embedding for that word.

```
'la': array([-6.18250e-03, -9.43867e-04, -8.82648e-03,  3.24623e-02,...
```

Load two dictionaries mapping the English to French words

- A training dictionary
- and a testing dictionary.

```
In [112]: # loading the english to french dictionaries
          en_fr_train = get_dict('en-fr.train.txt')
          print('The length of the English to French training dictionary is', len(en_fr_train))
          en_fr_test = get_dict('en-fr.test.txt')
          print('The length of the English to French test dictionary is', len(en_fr_train))

The length of the English to French training dictionary is 5000
The length of the English to French test dictionary is 5000
```

Looking at the English French dictionary

- `en_fr_train` is a dictionary where the key is the English word and the value is the French translation of that English word.

```
{'the': 'la',
 'and': 'et',
 'was': 'était',
 'for': 'pour',
```

- `en_fr_test` is similar to `en_fr_train`, but is a test set. We won't look at it until we get to testing.

2.2 1.1 Generate embedding and transform matrices

Exercise 01: Translating English dictionary to French by using embeddings

You will now implement a function `get_matrices`, which takes the loaded data and returns matrices `X` and `Y`.

Inputs: - `en_fr` : English to French dictionary - `en_embeddings` : English to embeddings dictionary - `fr_embeddings` : French to embeddings dictionary

Returns: - Matrix `X` and matrix `Y`, where each row in X is the word embedding for an english word, and the same row in Y is the word embedding for the French version of that English word.

Figure 2

Use the `en_fr` dictionary to ensure that the ith row in the `X` matrix corresponds to the ith row in the `Y` matrix.

Instructions: Complete the function `get_matrices()`: * Iterate over English words in `en_fr` dictionary. * Check if the word have both English and French embedding.

Hints

Sets are useful data structures that can be used to check if an item is a member of a group.

You can get words which are embedded into the language by using keys method.

Keep vectors in `X` and `Y` sorted in list. You can use np.vstack() to merge them into the numpy matrix.

numpy.vstack stacks the items in a list as rows in a matrix.

```
In [124]: # UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          def get_matrices(en_fr, french_vecs, english_vecs):
              """
              Input:
                  en_fr: English to French dictionary
                  french_vecs: French words to their corresponding word embeddings.
                  english_vecs: English words to their corresponding word embeddings.
              Output:
                  X: a matrix where the columns are the English embeddings.
                  Y: a matrix where the columns correspong to the French embeddings.
                  R: the projection matrix that minimizes the F norm ||X R -Y||^2.
              """

              ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###

              # X_l and Y_l are lists of the english and french word embeddings
              X_l = []
              Y_l = []

              # get the english words (the keys in the dictionary) and store in a set()
              #english_set = set(english_vecs.keys())
              english_set = english_vecs.keys()

              # get the french words (keys in the dictionary) and store in a set()
              french_set = french_vecs.keys()
```

```
    # store the french words that are part of the english-french dictionary (these a
    french_words = set(en_fr.values())

    # loop through all english, french word pairs in the english french dictionary
    for en_word, fr_word in en_fr.items():

        # check that the french word has an embedding and that the english word has
        if fr_word in french_set and en_word in english_set:

            # get the english embedding
            en_vec = english_vecs[en_word]

            # get the french embedding
            fr_vec = french_vecs[fr_word]

            # add the english embedding to the list
            X_l.append(en_vec)

            # add the french embedding to the list
            Y_l.append(fr_vec)

    # stack the vectors of X_l into a matrix X
    #X = np.stack(X_l, axis=1).shape# along row
    X = np.vstack(X_l)

    # stack the vectors of Y_l into a matrix Y
    #Y = np.stack(Y_l, axis=1).shape # along row
    Y = np.vstack(Y_l)

    ### END CODE HERE ###

    return X, Y
```

Now we will use function `get_matrices()` to obtain sets `X_train` and `Y_train` of English and French word embeddings into the corresponding vector space models.

In [125]:
```
# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so 

# getting the training set:
X_train, Y_train = get_matrices(
    en_fr_train, fr_embeddings_subset, en_embeddings_subset)
#print('X_train', X_train)
#print('Y_train', Y_train)
```

3 2. Translations

Figure 1

Write a program that translates English words to French words using word embeddings and vector space models.

2.1 Translation as linear transformation of embeddings

Given dictionaries of English and French word embeddings you will create a transformation matrix `R` * Given an English word embedding, $\mathbf{e}$, you can multiply $\mathbf{eR}$ to get a new word embedding $\mathbf{f}$. * Both $\mathbf{e}$ and $\mathbf{f}$ are row vectors. * You can then compute the nearest neighbors to `f` in the french embeddings and recommend the word that is most similar to the transformed word embedding.

3.0.1 Describing translation as the minimization problem

Find a matrix `R` that minimizes the following equation.

$$\arg\min_{\mathbf{R}} \|\mathbf{XR} - \mathbf{Y}\|_F \tag{1}$$

3.0.2 Frobenius norm

The Frobenius norm of a matrix A (assuming it is of dimension m, n) is defined as the square root of the sum of the absolute squares of its elements:

$$\|\mathbf{A}\|_F \equiv \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} \tag{2}$$

3.0.3 Actual loss function

In the real world applications, the Frobenius norm loss:

$$\|\mathbf{XR} - \mathbf{Y}\|_F$$

is often replaced by it's squared value divided by m:

$$\frac{1}{m} \|\mathbf{XR} - \mathbf{Y}\|_F^2$$

where m is the number of examples (rows in $\mathbf{X}$).

- The same R is found when using this loss function versus the original Frobenius norm.
- The reason for taking the square is that it's easier to compute the gradient of the squared Frobenius.
- The reason for dividing by m is that we're more interested in the average loss per embedding than the loss for the entire training set.
  - The loss for all training set increases with more words (training examples), so taking the average helps us to track the average loss regardless of the size of the training set.

[Optional] Detailed explanation why we use norm squared instead of the norm: Click for optional details

The norm is always nonnegative (we're summing up absolute values), and so is the square.

When we take the square of all non-negative (positive or zero) numbers, the order of the data is preserved.

For example, if 3 > 2, 3^2 > 2^2

Using the norm or squared norm in gradient descent results in the same location of the minimum.

Squaring cancels the square root in the Frobenius norm formula. Because of the chain rule, we would have to do more calculations if we had a square root in our expression for summation.

Dividing the function value by the positive number doesn't change the optimum of the function, for the same reason as described above.

We're interested in transforming English embedding into the French. Thus, it is more important to measure average loss per embedding than the loss for the entire dictionary (which increases as the number of words in the dictionary increases).

</p>

3.0.4 Exercise 02: Implementing translation mechanism described in this section.

Step 1: Computing the loss

- The loss function will be squared Frobenoius norm of the difference between matrix and its approximation, divided by the number of training examples m.
- Its formula is:
$$L(X, Y, R) = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} \left(a_{ij} \right)^2$$

where a_{ij} is value in ith row and jth column of the matrix $\mathbf{XR} - \mathbf{Y}$.

Instructions: complete the `compute_loss()` function

- Compute the approximation of `Y` by matrix multiplying `X` and `R`
- Compute difference `XR - Y`
- Compute the squared Frobenius norm of the difference and divide it by m.

Hints

Useful functions: Numpy dot , Numpy sum, Numpy square, Numpy norm

Be careful about which operation is elementwise and which operation is a matrix multiplication.

Try to use matrix operations instead of the numpy norm function. If you choose to use norm function, take care of extra arguments and that it's returning loss squared, and not the loss itself.

```
In [126]: # UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          def compute_loss(X, Y, R):
              '''
              Inputs:
                  X: a matrix of dimension (m,n) where the columns are the English embeddings.
```

```
        Y: a matrix of dimension (m,n) where the columns correspong to the French em
        R: a matrix of dimension (n,n) - transformation matrix from English to Frenc
    Outputs:
        L: a matrix of dimension (m,n) - the value of the loss function for given X,
    '''
    ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
    # m is the number of rows in X
    m = len(X)
    #print('m', m)

    # diff is XR - Y
    diff = np.dot(X, R)- Y

    # diff_squared is the element-wise square of the difference
    diff_squared = np.square(diff)

    # sum_diff_squared is the sum of the squared elements
    sum_diff_squared = np.sum(diff_squared)

    # loss is the sum_diff_squard divided by the number of examples (m)
    loss = sum_diff_squared/m
    ### END CODE HERE ###
    return loss
```

3.0.5 Exercise 03

3.0.6 Step 2: Computing the gradient of loss in respect to transform matrix R

- Calculate the gradient of the loss with respect to transform matrix `R`.
- The gradient is a matrix that encodes how much a small change in `R` affect the change in the loss function.
- The gradient gives us the direction in which we should decrease `R` to minimize the loss.
- m is the number of training examples (number of rows in X).
- The formula for the gradient of the loss function $(,,)$ is:

$$\frac{d}{dR}(,,) = \frac{d}{dR}\Big(\frac{1}{m}\|XR - Y\|_F^2\Big) = \frac{2}{m}X^T(XR - Y)$$

Instructions: Complete the `compute_gradient` function below.
Hints
Transposing in numpy
Finding out the dimensions of matrices in numpy
Remember to use numpy.dot for matrix multiplication

In [127]:
```
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def compute_gradient(X, Y, R):
    '''
    Inputs:
        X: a matrix of dimension (m,n) where the columns are the English embeddings.
```

```
    Y: a matrix of dimension (m,n) where the columns correspong to the French em
    R: a matrix of dimension (n,n) - transformation matrix from English to Frenc
Outputs:
    g: a matrix of dimension (n,n) - gradient of the loss function L for given X
'''
### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
# m is the number of rows in X
m = len(X)

# gradient is X^T(XR - Y) * 2/m
gradient =  (2/m)*np.dot(np.transpose(X), (np.dot(X,R)-Y))
### END CODE HERE ###
return gradient
```

3.0.7 Step 3: Finding the optimal R with gradient descent algorithm

Gradient descent Gradient descent is an iterative algorithm which is used in searching for the optimum of the function. * Earlier, we've mentioned that the gradient of the loss with respect to the matrix encodes how much a tiny change in some coordinate of that matrix affect the change of loss function. * Gradient descent uses that information to iteratively change matrix `R` until we reach a point where the loss is minimized.

Training with a fixed number of iterations Most of the time we iterate for a fixed number of training steps rather than iterating until the loss falls below a threshold.

OPTIONAL: explanation for fixed number of iterations click here for detailed discussion

You cannot rely on training loss getting low – what you really want is the validation loss to go down, or validation accuracy to go up. And indeed - in some cases people train until validation accuracy reaches a threshold, or – commonly known as "early stopping" – until the validation accuracy starts to go down, which is a sign of over-fitting.

Why not always do "early stopping"? Well, mostly because well-regularized models on larger data-sets never stop improving. Especially in NLP, you can often continue training for months and the model will continue getting slightly and slightly better. This is also the reason why it's hard to just stop at a threshold – unless there's an external customer setting the threshold, why stop, where do you put the threshold?

Stopping after a certain number of steps has the advantage that you know how long your training will take - so you can keep some sanity and not train for months. You can then try to get the best performance within this time budget. Another advantage is that you can fix your learning rate schedule – e.g., lower the learning rate at 10% before finish, and then again more at 1% before finishing. Such learning rate schedules help a lot, but are harder to do if you don't know how long you're training.

Pseudocode: 1. Calculate gradient g of the loss with respect to the matrix R. 2. Update R with the formula:

$$R_{\text{new}} = R_{\text{old}} - \alpha g$$

Where α is the learning rate, which is a scalar.

Learning rate

- The learning rate or "step size" α is a coefficient which decides how much we want to change R in each step.
- If we change R too much, we could skip the optimum by taking too large of a step.
- If we make only small changes to R, we will need many steps to reach the optimum.
- Learning rate α is used to control those changes.
- Values of α are chosen depending on the problem, and we'll use `learning_rate`$= 0.0003$ as the default value for our algorithm.

3.0.8 Exercise 04

Instructions: Implement `align_embeddings()` Hints
Use the 'compute_gradient()' function to get the gradient in each step

In [128]:
```python
# UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def align_embeddings(X, Y, train_steps=100, learning_rate=0.0003):
    '''
    Inputs:
        X: a matrix of dimension (m,n) where the columns are the English embeddings.
        Y: a matrix of dimension (m,n) where the columns correspong to the French em
        train_steps: positive int - describes how many steps will gradient descent a
        learning_rate: positive float - describes how big steps will  gradient desce
    Outputs:
        R: a matrix of dimension (n,n) - the projection matrix that minimizes the F 
    '''
    np.random.seed(129)


    # the number of columns in X is the number of dimensions for a word vector (e.g.
    # R is a square matrix with length equal to the number of dimensions in the  wor
    num_rows, num_cols = X.shape
    R = np.random.rand(num_cols, num_cols)

    for i in range(train_steps):
        if i % 25 == 0:
            print(f"loss at iteration {i} is: {compute_loss(X, Y, R):.4f}")

        ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###

        # use the function that you defined to compute the gradient
        #gradient = None
        gradient = compute_gradient(X, Y, R)

        # update R by subtracting the learning rate times gradient
        R -= np.dot(learning_rate,gradient)
        ### END CODE HERE ###
    return R
```

```
In [129]: # UNQ_C6 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          # You do not have to input any code in this cell, but it is relevant to grading, so p

          # Testing your implementation.
          np.random.seed(129)
          m = 10
          n = 5
          X = np.random.rand(m, n)
          Y = np.random.rand(m, n) * .1
          R = align_embeddings(X, Y)
```

```
loss at iteration 0 is: 3.7242
loss at iteration 25 is: 3.6283
loss at iteration 50 is: 3.5350
loss at iteration 75 is: 3.4442
```

Expected Output:

```
loss at iteration 0 is: 3.7242
loss at iteration 25 is: 3.6283
loss at iteration 50 is: 3.5350
loss at iteration 75 is: 3.4442
```

3.1 Calculate transformation matrix R

Using those the training set, find the transformation matrix **R** by calling the function `align_embeddings()`.

NOTE: The code cell below will take a few minutes to fully execute (~3 mins)

```
In [130]: # UNQ_C7 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          # You do not have to input any code in this cell, but it is relevant to grading, so p
          R_train = align_embeddings(X_train, Y_train, train_steps=400, learning_rate=0.8)
```

```
loss at iteration 0 is: 963.0146
loss at iteration 25 is: 97.8292
loss at iteration 50 is: 26.8329
loss at iteration 75 is: 9.7893
loss at iteration 100 is: 4.3776
loss at iteration 125 is: 2.3281
loss at iteration 150 is: 1.4480
loss at iteration 175 is: 1.0338
loss at iteration 200 is: 0.8251
loss at iteration 225 is: 0.7145
loss at iteration 250 is: 0.6534
loss at iteration 275 is: 0.6185
loss at iteration 300 is: 0.5981
loss at iteration 325 is: 0.5858
loss at iteration 350 is: 0.5782
loss at iteration 375 is: 0.5735
```

Expected Output

```
loss at iteration 0 is: 963.0146
loss at iteration 25 is: 97.8292
loss at iteration 50 is: 26.8329
loss at iteration 75 is: 9.7893
loss at iteration 100 is: 4.3776
loss at iteration 125 is: 2.3281
loss at iteration 150 is: 1.4480
loss at iteration 175 is: 1.0338
loss at iteration 200 is: 0.8251
loss at iteration 225 is: 0.7145
loss at iteration 250 is: 0.6534
loss at iteration 275 is: 0.6185
loss at iteration 300 is: 0.5981
loss at iteration 325 is: 0.5858
loss at iteration 350 is: 0.5782
loss at iteration 375 is: 0.5735
```

3.2 2.2 Testing the translation

3.2.1 k-Nearest neighbors algorithm

k-Nearest neighbors algorithm * k-NN is a method which takes a vector as input and finds the other vectors in the dataset that are closest to it. * The 'k' is the number of "nearest neighbors" to find (e.g. k=2 finds the closest two neighbors).

3.2.2 Searching for the translation embedding

Since we're approximating the translation function from English to French embeddings by a linear transformation matrix **R**, most of the time we won't get the exact embedding of a French word when we transform embedding **e** of some particular English word into the French embedding space. * This is where k-NN becomes really useful! By using 1-NN with **eR** as input, we can search for an embedding **f** (as a row) in the matrix **Y** which is the closest to the transformed vector **eR**

3.2.3 Cosine similarity

Cosine similarity between vectors u and v calculated as the cosine of the angle between them. The formula is

$$\cos(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|}$$

- $\cos(u, v) = 1$ when u and v lie on the same line and have the same direction.
- $\cos(u, v)$ is -1 when they have exactly opposite directions.
- $\cos(u, v)$ is 0 when the vectors are orthogonal (perpendicular) to each other.

Note: Distance and similarity are pretty much opposite things.

- We can obtain distance metric from cosine similarity, but the cosine similarity can't be used directly as the distance metric.
- When the cosine similarity increases (towards 1), the "distance" between the two vectors decreases (towards 0).
- We can define the cosine distance between u and v as

$$d_{\cos}(u,v) = 1 - \cos(u,v)$$

Exercise 05: Complete the function `nearest_neighbor()`

Inputs: * Vector `v`, * A set of possible nearest neighbors `candidates` * `k` nearest neighbors to find. * The distance metric should be based on cosine similarity. * `cosine_similarity` function is already implemented and imported for you. It's arguments are two vectors and it returns the cosine of the angle between them. * Iterate over rows in `candidates`, and save the result of similarities between current row and vector `v` in a python list. Take care that similarities are in the same order as row vectors of `candidates`. * Now you can use numpy argsort to sort the indices for the rows of `candidates`.

Hints

numpy.argsort sorts values from most negative to most positive (smallest to largest)

The candidates that are nearest to 'v' should have the highest cosine similarity

To get the last element of a list 'tmp', the notation is tmp[-1:]

In [158]:
```
# UNQ_C8 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def nearest_neighbor(v, candidates, k=1):
    """
    Input:
      - v, the vector you are going find the nearest neighbor for
      - candidates: a set of vectors where we will find the neighbors
      - k: top k nearest neighbors to find
    Output:
      - k_idx: the indices of the top k closest vectors in sorted form
    """
    ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
    similarity_l = []

    # for each candidate vector...
    for row in candidates:
        # get the cosine similarity
        cos_similarity = cosine_similarity(v,row)
        #print('row cos_similarity', row, cos_similarity)

        # append the similarity to the list
        similarity_l.append(cos_similarity)

    # sort the similarity list and get the indices of the sorted list
    sorted_ids = np.argsort(similarity_l)
```

```
        # get the indices of the k most similar candidate vectors
        #[::-1] reverses the array returned by argsort() and [:n] gives that last n elem
        k_idx = sorted_ids[-k:]


        ### END CODE HERE ###
        return k_idx
```

In [159]:
```
# UNQ_C9 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so 

# Test your implementation:
v = np.array([1, 0, 1])
candidates = np.array([[1, 0, 5], [-2, 5, 3], [2, 0, 1], [6, -9, 5], [9, 9, 9]])
print(candidates[nearest_neighbor(v, candidates, 3)])
```

```
[[9 9 9]
 [1 0 5]
 [2 0 1]]
```

Expected Output:
`[[9 9 9] [1 0 5] [2 0 1]]`

3.2.4 Test your translation and compute its accuracy

Exercise 06: Complete the function `test_vocabulary` which takes in English embedding matrix X, French embedding matrix Y and the R matrix and returns the accuracy of translations from X to Y by R.

- Iterate over transformed English word embeddings and check if the closest French word vector belongs to French word that is the actual translation.
- Obtain an index of the closest French embedding by using `nearest_neighbor` (with argument `k=1`), and compare it to the index of the English embedding you have just transformed.
- Keep track of the number of times you get the correct translation.
- Calculate accuracy as

$$\text{accuracy} = \frac{\#(\text{correct predictions})}{\#(\text{total predictions})}$$

In [160]:
```
# UNQ_C10 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def test_vocabulary(X, Y, R):
    '''
    Input:
        X: a matrix where the columns are the English embeddings.
        Y: a matrix where the columns correspong to the French embeddings.
        R: the transform matrix which translates word embeddings from
        English to French word vector space.
    Output:
```

```
        accuracy: for the English to French capitals
    '''

    ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
    # The prediction is X times R
    pred = np.dot(X, R)

    # initialize the number correct to zero
    num_correct = 0

    # loop through each row in pred (each transformed embedding)
    for i in range(len(pred)):
        # get the index of the nearest neighbor of pred at row 'i'; also pass in the
        pred_idx = nearest_neighbor(pred[i], Y, 1)

        # if the index of the nearest neighbor equals the row of i... \
        if pred_idx == i:
            # increment the number correct by 1.
            num_correct += 1

    # accuracy is the number correct divided by the number of rows in 'pred' (also n
    accuracy = num_correct/ len(pred)

    ### END CODE HERE ###

    return accuracy
```

Let's see how is your translation mechanism working on the unseen data:

```
In [161]: X_val, Y_val = get_matrices(en_fr_test, fr_embeddings_subset, en_embeddings_subset)
```

```
In [162]: # UNQ_C11 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          # You do not have to input any code in this cell, but it is relevant to grading, so p

          acc = test_vocabulary(X_val, Y_val, R_train)  # this might take a minute or two
          print(f"accuracy on test set is {acc:.3f}")
```

```
accuracy on test set is 0.557
```

Expected Output:

```
0.557
```

You managed to translate words from one language to another language without ever seing them with almost 56% accuracy by using some basic linear algebra and learning a mapping of words from one language to another!

4 3. LSH and document search

In this part of the assignment, you will implement a more efficient version of k-nearest neighbors using locality sensitive hashing. You will then apply this to document search.

- Process the tweets and represent each tweet as a vector (represent a document with a vector embedding).
- Use locality sensitive hashing and k nearest neighbors to find tweets that are similar to a given tweet.

```
In [163]: # get the positive and negative tweets
          all_positive_tweets = twitter_samples.strings('positive_tweets.json')
          all_negative_tweets = twitter_samples.strings('negative_tweets.json')
          all_tweets = all_positive_tweets + all_negative_tweets
```

4.0.1 3.1 Getting the document embeddings

Bag-of-words (BOW) document models Text documents are sequences of words. * The ordering of words makes a difference. For example, sentences "Apple pie is better than pepperoni pizza." and "Pepperoni pizza is better than apple pie" have opposite meanings due to the word ordering. * However, for some applications, ignoring the order of words can allow us to train an efficient and still effective model. * This approach is called Bag-of-words document model.

Document embeddings

- Document embedding is created by summing up the embeddings of all words in the document.
- If we don't know the embedding of some word, we can ignore that word.

Exercise 07: Complete the `get_document_embedding()` function. * The function `get_document_embedding()` encodes entire document as a "document" embedding. * It takes in a docoument (as a string) and a dictionary, `en_embeddings` * It processes the document, and looks up the corresponding embedding of each word. * It then sums them up and returns the sum of all word vectors of that processed tweet.

Hints

You can handle missing words easier by using the `get()` method of the python dictionary instead of the bracket notation (i.e. "[ ]"). See more about it here

The default value for missing word should be the zero vector. Numpy will broadcast simple 0 scalar into a vector of zeros during the summation.

Alternatively, skip the addition if a word is not in the dictonary.

You can use your `process_tweet()` function which allows you to process the tweet. The function just takes in a tweet and returns a list of words.

```
In [164]: # UNQ_C12 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          def get_document_embedding(tweet, en_embeddings):
              '''
              Input:
                  - tweet: a string
                  - en_embeddings: a dictionary of word embeddings
```

```
    Output:
        - doc_embedding: sum of all word embeddings in the tweet
    '''
    doc_embedding = np.zeros(300)

    ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
    # process the document into a list of words (process the tweet)
    processed_doc = process_tweet(tweet)
    for word in processed_doc:
        # add the word embedding to the running total for the document embedding
        doc_embedding += en_embeddings.get(word,0)
    ### END CODE HERE ###
    return doc_embedding
```

In [165]:
```
# UNQ_C13 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so p

# testing your function
custom_tweet = "RT @Twitter @chapagain Hello There! Have a great day. :) #good #morni
tweet_embedding = get_document_embedding(custom_tweet, en_embeddings_subset)
tweet_embedding[-5:]
```

Out[165]:
```
array([-0.00268555, -0.15378189, -0.55761719, -0.07216644, -0.32263184])
```

Expected output:

```
array([-0.00268555, -0.15378189, -0.55761719, -0.07216644, -0.32263184])
```

4.0.2 Exercise 08

Store all document vectors into a dictionary Now, let's store all the tweet embeddings into a dictionary. Implement `get_document_vecs()`

In [166]:
```
# UNQ_C14 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def get_document_vecs(all_docs, en_embeddings):
    '''
    Input:
        - all_docs: list of strings - all tweets in our dataset.
        - en_embeddings: dictionary with words as the keys and their embeddings as th
    Output:
        - document_vec_matrix: matrix of tweet embeddings.
        - ind2Doc_dict: dictionary with indices of tweets in vecs as keys and their e
    '''

    # the dictionary's key is an index (integer) that identifies a specific tweet
    # the value is the document embedding for that document
    ind2Doc_dict = {}

    # this is list that will store the document vectors
```

```
    document_vec_l = []

    for i, doc in enumerate(all_docs):
        #print('i', i)
        #print('doc', doc)

        ### START CODE HERE (REPLACE INSTANCES OF 'None' with your code) ###
        # get the document embedding of the tweet
        doc_embedding = get_document_embedding(doc, en_embeddings)

        # save the document embedding into the ind2Tweet dictionary at index i
        ind2Doc_dict[i] = doc_embedding

        # append the document embedding to the list of document vectors
        document_vec_l.append(ind2Doc_dict[i])

        ### END CODE HERE ###

    # convert the list of document vectors into a 2D array (each row is a document v
    document_vec_matrix = np.vstack(document_vec_l)

    return document_vec_matrix, ind2Doc_dict
```

```
In [167]: document_vecs, ind2Tweet = get_document_vecs(all_tweets, en_embeddings_subset)
```

```
In [168]: # UNQ_C15 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          # You do not have to input any code in this cell, but it is relevant to grading, so p

          print(f"length of dictionary {len(ind2Tweet)}")
          print(f"shape of document_vecs {document_vecs.shape}")
```

```
length of dictionary 10000
shape of document_vecs (10000, 300)
```

Expected Output

```
length of dictionary 10000
shape of document_vecs (10000, 300)
```

4.1 3.2 Looking up the tweets

Now you have a vector of dimension (m,d) where `m` is the number of tweets (10,000) and `d` is the dimension of the embeddings (300). Now you will input a tweet, and use cosine similarity to see which tweet in our corpus is similar to your tweet.

```
In [169]: my_tweet = 'i am sad'
          process_tweet(my_tweet)
          tweet_embedding = get_document_embedding(my_tweet, en_embeddings_subset)
```

```
In [170]: # UNQ_C16 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
          # You do not have to input any code in this cell, but it is relevant to grading, so 

          # this gives you a similar tweet as your input.
          # this implementation is vectorized...
          idx = np.argmax(cosine_similarity(document_vecs, tweet_embedding))
          print(all_tweets[idx])
```

```
@zoeeylim sad sad sad kid :( it's ok I help you watch the match HAHAHAHAHA
```

Expected Output

```
@zoeeylim sad sad sad kid :( it's ok I help you watch the match HAHAHAHAHA
```

4.2 3.3 Finding the most similar tweets with LSH

You will now implement locality sensitive hashing (LSH) to identify the most similar tweet. * Instead of looking at all 10,000 vectors, you can just search a subset to find its nearest neighbors.

Let's say your data points are plotted like this:

Figure 3

You can divide the vector space into regions and search within one region for nearest neighbors of a given vector.

Figure 4

```
In [171]: N_VECS = len(all_tweets)       # This many vectors.
          N_DIMS = len(ind2Tweet[1])     # Vector dimensionality.
          print(f"Number of vectors is {N_VECS} and each has {N_DIMS} dimensions.")
```

```
Number of vectors is 10000 and each has 300 dimensions.
```

Choosing the number of planes

- Each plane divides the space to 2 parts.
- So n planes divide the space into 2^n hash buckets.
- We want to organize 10,000 document vectors into buckets so that every bucket has about 16 vectors.
- For that we need $\frac{10000}{16} = 625$ buckets.
- We're interested in n, number of planes, so that $2^n = 625$. Now, we can calculate $n = \log_2 625 = 9.29 \approx 10$.

```
In [172]: # The number of planes. We use log2(625) to have ~16 vectors/bucket.
          N_PLANES = 10
          # Number of times to repeat the hashing to improve the search.
          N_UNIVERSES = 25
```

4.3 3.4 Getting the hash number for a vector

For each vector, we need to get a unique number associated to that vector in order to assign it to a "hash bucket".